2011 年度 修士論文

ハイブリッド制約言語 HydLa の REDUCE を用いた高信頼実行処理系

提出日　：　2012 年 1 月 27 日
指導　　：　上田　和紀 教授

早稲田大学大学院 基幹理工学研究科
情報理工学専攻

学籍番号：5110B075-1
高田　賢士郎

概要

現代社会における様々な情報システムにおいては，安全性の保証などが期待されるものが数多く存在するが，その複雑さが増すのに伴い，システムに求められる性質の確実な検証は困難なものとなる．システムの性質の検証を，計算機を用いたシミュレーションによって厳密におこなうには誤差の無い計算が不可欠であり，かつ，それを簡潔な記述により実現できる手法が求められている．

システムの挙動や現象を，時間経過に伴った状態の連続変化と離散変化と見なして扱うハイブリッドシステムという概念は，物理学，電気工学，制御工学など幅広い分野のモデルに対し，同じ枠組みによって統一的に適用することができる．ハイブリッドシステムの記述形式としてハイブリッドオートマトンや Hybrid CC などを挙げることができるが，複雑なモデルの場合取りうる状態をすべて過不足なく記述しなければならず，また，計算の精度が保証されていないために厳密なシミュレーションがおこなえない．

ハイブリッド制約言語 HydLa は，ハイブリッドシステムの記述，実行，検証を目的としており，制約概念と制約階層に基づいて対象のモデル化がおこなえる．Hy dLa の処理系である Hyrose は，HydLa で記述された入力プログラムを解釈し，そのモデルの挙動をシミュレーション実行し，検証をおこなう．

Hyrose における制約求解ソルバの 1 つである REDUCE Constraint Solver は数式処理システムである REDUCE を用いることで，シミュレーション実行において必要となる種々の計算を行っている．Hyrose ではシミュレーションをおこなう際に，ある連続変化が継続している状態から何らかの離散変化が起きる時刻を制約処理によって求めることでシミュレーション時刻を先に進めており，その際に 2 値の厳密な大小判定が必要となる．

しかし，REDUCE による計算処理においては，無理数に関する厳密な大小比較をおこなうことができないという問題を有している．そこで，無理数を初めとする，数式処理を用いた大小比較のおこなえない定数を対象として，無限長整数によって表された有理数の区間値に変換し，区間演算を施すことによって，定数の符号判定処理を実現する手法の提案をおこなった．また，これらの提案手法を REDUCE により実装し，REDUCE Constraint Solver への導入をおこなった．

本研究により，REDUCE Constraint Solver は HydLa で記述されたモデルに対して，必要に応じて区間演算を交えつつ計算する高信頼な数式処理シミュレーションをおこなうことが可能となった．また，正しいシミュレーション結果が得られていることを，いくつかの例題をもとに確認した．

目次

図目次

表目次

第 1 章

はじめに

1.1 研究の背景と目的

現代社会における様々な情報システムにおいては，安全性の保証などが期待されるものが数多く存在するが，その複雑さが増すのに伴い，システムに求められる性質の確実な検証は困難なものとなる．システムの性質の検証を，計算機を用いたシミュレーションによって厳密におこなうには誤差の無い計算が不可欠であり，かつ，それを簡潔な記述により実現できる手法が求められている．

システムの挙動や現象を，時間経過に伴った状態の連続変化と離散変化と見なして扱うハイブリッドシステムという概念は，物理学，電気工学，制御工学など幅広い分野のモデルに対し，同じ枠組みによって統一的に適用することができる．ハイブリッドシステムの記述形式としてハイブリッドオートマトンや Hybrid CC などを挙げることができるが，複雑なモデルの場合取りうる状態をすべて過不足なく記述しなければならず，また，計算の精度が保証されていないために厳密なシミュレーションがおこなえない．

ハイブリッド制約言語 HydLa[7] は，ハイブリッドシステムの記述，実行，検証を目的としており，制約概念と制約階層 [1] に基づいて対象のモデル化がおこなえる．Hy dLa の処理系である Hyrose は，HydLa で記述された入力プログラムを解釈し，そのモデルの挙動をシミュレーション実行し，検証をおこなう．

Hyrose における制約求解ソルバの 1 つである REDUCE Constraint Solver は数式処理エンジンである REDUCE を用いることで，シミュレーション実行において必要となる種々の計算を行っている．Hyrose ではシミュレーションをおこなう際に，ある連続変化が継続している状態から何らかの離散変化が起きる時刻を制約処理によって求めることでシミュレーション時刻を先に進めており，その際に 2 値の厳密な大小判定が必要となる．

しかし，REDUCE による計算処理においては，無理数に関する厳密な大小比較をおこなうことができないという問題を有している．そこで，無理数を初めとする，数式処理を用いた大小比較のおこなえない定数を対象として，無限長整数によって表された有理数の区間値に変換し，区間演算を施すことによって，定数の符号判定処理を実現する手法の提案をおこなった．また，これらの提案手法を REDUCE により実装し，REDUCE Constraint Solver への導入をおこなった．

1.2 論文構成

本論文の構成は下記の通りである．

第 2 章では， HydLa 言語の概要についての解説をおこなう．第 3 章では，HydLa の処理系である Hyrose の実行アルゴリズムについて，解説をおこなう．第 4 章では，本研究で実装をおこなった Hyrose の数式処理制約求解系である REDUCE Constraint Solver (RCS) における制約処理手法について, 解説をおこなう．第 5 章では，REDUCE Constraint Solver（RCS）に導入した，区間演算を用いた符号判定法について解説をおこなう．第 6 章では， RCS の具体的な実装について述べ，各クラスの詳細とアルゴリズムの解説をおこなう．第 7 章では，具体的な例題をもとに，RCS と導入した区間演算符号判定法とが正しく機能していることを確認する．第 8 章では，本研究のまとめと，今後の課題について述べる．

第 2 章

ハイブリッド制約言語 HydLa

HydLa は，上田研究室 HydLa 班により開発された，ハイブリッドシステムモデリング言語である [8]．本章では HydLa の言語としての側面と処理系としての側面から解説を行う．

2.1 ハイブリッドシステム

ハイブリッドシステムとは，時間変化に伴って連続変化と離散変化が起きるようなシステムのことである．身近な例としては，力学における物体の衝突運動や摩擦力のかかる運動などがある．また，他にも電気の分野におけるスイッチを含む回路なども挙げられ，幅広い用途に応用が可能である．

2.2 HydLa 言語

2.2.1 概要

HydLa 言語には，その特徴として以下のようなものが挙げられる．

- 宣言型言語
  数学，論理学の記法を最大限利用した構文体系となっている．これにより，手続き型言語のように習得の手間をかけることなく記述が行える．
- 制約ベース
  モデルにおいて変数が満たすべき条件を，微分方程式や論理記号から成る制約によって記述を行う．HydLa プログラムにおける変数は実際は時刻 t に関する関数

変数であり，導関数を表す’や左極限値を表す-といった直感的な記法が提供されている．

- 制約階層
  それぞれの制約間に対して優先順位を設ける，制約階層 [1] を用いている．これにより，ある時点ごとの制約を過不足なく与えるといった必要はなく，最低限の記述のみで済むことになる．

このような特徴から，より直感的で簡潔な記述によってハイブリッドシステムをモデリングすることが可能となっている．このため，プログラミングやシミュレーションを専門としない技術者でも利用できる言語となることが期待されている．

2.2.2 tell 制約と ask 制約

HydLa では，2.2.1 節で述べた通り微分方程式と論理記号によって，制約を記述することでモデリングを行う．制約は大きく分けて 2 種類あり，tell 制約と ask 制約とが存在する．記号=>を含む制約を ask 制約といい，それ以外の制約を tell 制約という．ask 制約において，=>の左辺に記述される式はガード条件と呼ばれ，この条件を満たす（entail）場合に初めて右辺の式が評価される．このとき，右辺の式は新たな tell 制約として扱われ，この一連の流れを ask 制約の展開という．

また，tell 制約・ask 制約を問わず，制約の先頭に [] がつく場合がある．[] は論理記号の Always を表しており，制約の先頭についた場合，その制約が時刻 $t(\geq 0)$ に関して常に成り立つことを意味する．また，[] がつかない場合は時刻 $t = 0$ に関してのみその制約が成り立つことを意味する．

2.2.3 HydLa によるモデリング例

以下に，HydLa によるモデリングの例を示す．

これは，空中にある質点が自然落下し，地面でバウンドするモデルを HydLa によって表したものである．このモデルにおいて，y は質点の高さを表している．また，y’ は y の時間に関する微分，つまり y 方向に関する速さを表している．1 行目の INIT は tell 制約であり，[] はついていないので，時刻 $t = 0$ に関してのみ成り立つ．この記述はつまり，ボールの高さ y とその落下速度 y’ の初期値を定めていることになる．2 行目の FALL も tell 制約であるが，[] がついているので，時刻 t に関して常に成り立つ．なお，y’’ とは，y 方向の速さを時間に関して微分したものであるので，y 方向の加速度を表している

```
INIT   <=> y=10 /\ y'=0.
FALL   <=> [](y'' = -10).
BOUNCE <=> [](y = 0 => y' = -(4/5) * y'-).

INIT, FALL << BOUNCE.
```

図 2.1 HydLa プログラムの例：bouncing_particle

ことになる．この記述はつまり，質点は常にある加速度（重力加速度 G．この例ではその値を 10 としている）のもと，落下するということを表している．3 行目の BOUNCE は制約中に=>が含まれるので ask 制約であり，また，これも [] がついているので，時刻 t に関して常に成り立つ．ここでは，=>の左辺で指定された条件 y- = 0 を満たす場合のみ右辺の式 y' = -(4/5) * y'-が成り立つことを指定している．この記述はつまり，質点の高さ y が 0 になったときは質点が地面に衝突してバウンドするので，その速さ y' が逆向きになることを表している．その際，反発係数（この例では 0.8 としている）を考慮しているので，バウンドするたびにその絶対値は小さくなる．

最後の行では，ここまでで記述したそれぞれの制約間の強さ（優先順位）を指定しており，これにより制約階層が形成されている．FALL と BOUNCE という 2 つの制約は，そのどちらもが同時に成り立つということはない（落下しながらバウンドすることはない）．そこで，BOUNCE を FALL よりも強い制約とすることで，ask 制約のガード条件が満たされるときだけバウンドし，それ以外は落下するという挙動を簡潔に記述することが可能となっている．また，初期値の指定を行う制約 INIT に関しては，特に優先順位を指定することなく合成を行っている．

2.2.4 HydLa の解軌道

HydLa の解軌道は，時刻 t に関する各関数変数の値の変遷によって表され，その取りうる値は実数領域のみを考える．ハイブリッドシステムにおいては，ある時刻において 1 つの関数変数が同時に異なる値を取ることがあるが（ハイブリッド時刻）[?]，HydLa における解軌道ではそのような挙動を許していない．

2.2.5 HydLa における連続性

HydLa における各関数変数の連続性に関しては，いまだに定式化がなされていない．しかし，ユーザがプログラムに記述した制約とは別に，新たに追加して考えるべき制約であると見なすことができる．HydLa において最低でも必要であると考えられる，望まれる性質としては以下の通りとなっている．

1. 微分制約よりも強いレベルにある
2. BOUNCE のような，離散変化が起きるきっかけになる制約よりも弱いレベルにある
3. 左連続性と右連続性とが別々に存在する
4. 同じ時刻において，左右どちらの連続性も存在しない場合もある

上記 4 については，パルス関数のような解軌道において起きる．

2.2.6 HydLa における全解探索

```
INIT   <=> 9<=y /\ y<=11 /\ y'=0.
FALL   <=> [](y'' = -10).
BOUNCE <=> [](y = 0 => y' = -(4/5) * y'-).

INIT, FALL << BOUNCE.
```

図 2.2 初期値に不確実値を持つ HydLa プログラムの例：bouncing_particle_interval

HydLa では，図 2.2 に示すプログラムのように，不等式を含むようなモデルの記述も許している．これは，先述の地面でバウンドする質点のモデルにおいて，初期値が幅を持った不確実値である場合を表したモデルである．このようなモデルの場合，その解軌道は複数考えられることがある．HydLa はハイブリッドシステムの検証をおこなうことをその目的の 1 つとしているが，図 2.2 のようなモデルにも対応するには，1 つの解だけではなく，HydLa で記述されたモデルが与えうるすべての解軌道を求める全解探索 [7] が必要となる．また，不等式を含まないモデルにおいても，シミュレーションの途中において解軌道の分岐が発生する場合が考えられ [7]，この場合もやはり同様に，すべての解軌道を全

解探索する必要がある．

2.3 HydLa を対象としたツール

2.3.1 Hyrose

Hyrose は HydLa 言語の処理系であり，HydLa で記述されたプログラム（モデル）を入力としてそのシミュレーションをおこなう．実行時に打ちきり時刻を与えることで，その時刻までのシミュレーションをおこなう．

現在の Hyrose の構成概要を表したものを，図 2.3 に示す．

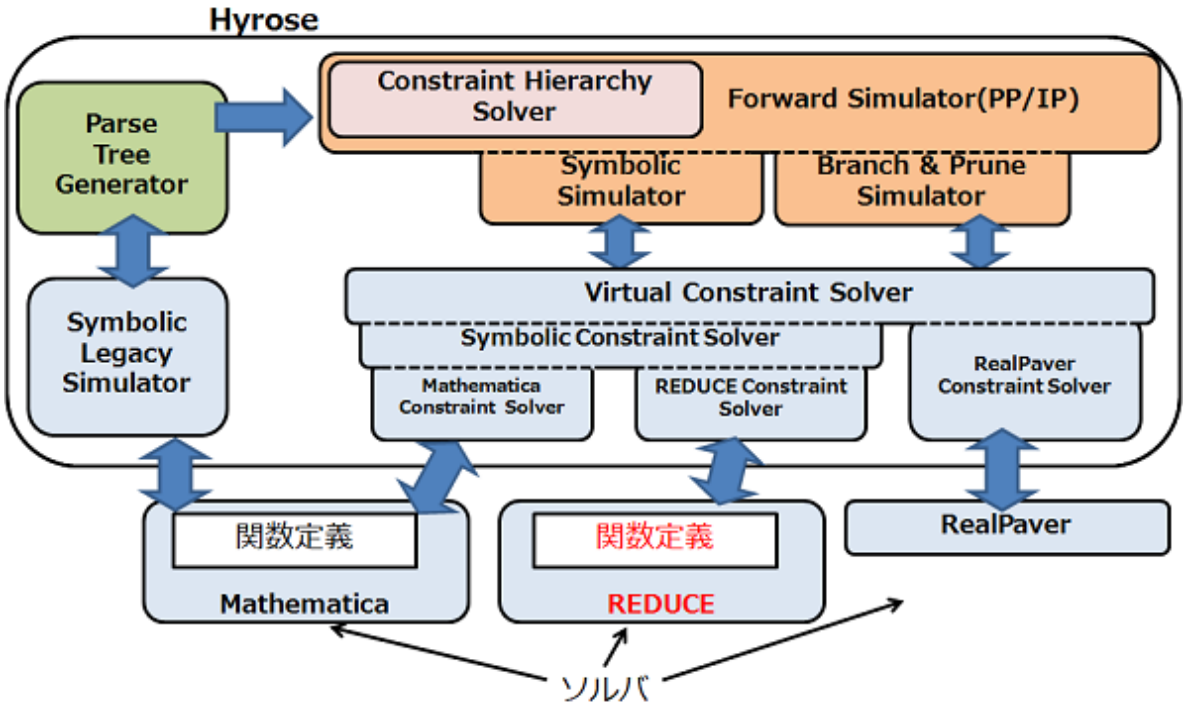

図 2.3 HydLa 処理系の構成図

Hyrose の処理としては，大きく分けてパース処理，制約階層処理，フェーズ切り替え処理，計算処理に分類される．これらのうち，パース処理，制約階層処理，フェーズ切り替え処理に関しては，Hyrose におけるフロントエンドにあたる C++ により実装されており，計算処理はバックエンドにあたる各ソルバ（計算機能を持つソフトウェア等）により実装されている．ソルバとしては数式処理に基づいて計算をおこなうものと区間演算に基

づいて計算をおこなうものとを採用している．数式処理ソルバとしては Mathematica[5], REDUCE[6] が，区間演算ソルバとしては RealPaver[3] が現時点では採用されている．

なお，これら処理の 1 つ 1 つはそれぞれ個別の機能モジュールとして実装がなされている．そのため，たとえばある 1 つの機能を別アプローチによって実現する必要が生じた際などにおいても，モジュールレベルで差し替えをおこなうことにより比較的容易に変更・拡張が可能な実装となっている．

2.3.2 HIDE

HIDE（HydLa IDE）は HydLa 言語の統合開発環境である．HIDE を用いることで，HydLa により記述したモデルのシミュレーションおよび検証等をより容易におこなうことができる．

第 3 章

Hyrose による HydLa の実行アルゴリズム

本章では HydLa の処理系である Hyrose の，数式処理に基づいたシミュレーション実行アルゴリズムについて，解説を行う．以下は [4] の内容に基づいて記述している．

3.1 パース処理

HydLa プログラムを入力として受け取った Hyrose は，まず HydLa プログラムのパース処理として，制約定義およびプログラム定義の展開をおこなう．展開は HydLa の構文定義に従っておこなわれ，構文定義に反する記述がプログラム内に認められた場合，その旨を表示した上で実行を終了する．展開の結果得られた構文木は，すべて等式や不等式および時相論理演算子 [] と論理演算子によって構成された形式になっている．

3.2 制約階層処理

パース処理が終わると，続いて制約階層処理をおこなう．Hyrose においては，HydLa プログラム中で与えられた複数の制約モジュールに対し，以下のような規則に基づいてシミュレーションをおこなうことが求められる．

1. 制約モジュール間に矛盾がない
2. なるべく多くの制約モジュールを満たす
3. 制約モジュール間の強弱関係の指定（制約階層）に従っている

Require: HydLa プログラム HP, シミュレーション終了時刻 $MaxT$
Ensure: HydLa プログラムの解軌道

```
1:  MS :=TopologicalSort(SolveCH(HP))
2:  T := 0
3:  S := true
4:  while T <_S MaxT do
5:    for M ∈ MS do
6:      (SS, MS_tmp) :=PP(S, M, MS, T)
7:      while |SS| > 1 do
8:        S :=GetElement(SS)
9:        (SS, MS_tmp) :=PP(S, M, MS, T)
10:     end while
11:     if |SS| = 1 then
12:       S :=GetElement(SS)
13:       MS := MS_tmp
14:       goto PPEnd
15:     end if
16:   end for
17:   break // 全ての制約モジュール集合が矛盾
18:   PPEnd:
19:   for M ∈ MS do
20:     (SS, T_tmp, MS_tmp) :=IP(S, M, MS, T, MaxT)
21:     while |SS| > 1 do
22:       S :=GetElement(SS)
23:       (SS, T_tmp, MS_tmp) :=IP(S, M, MS, T, MaxT)
24:     end while
25:     if |SS| = 1 then
26:       S :=GetElement(SS)
27:       MS := MS_tmp
28:       T := T_tmp
29:       goto IPEnd
30:     end if
31:   end for
32:   break // 全ての制約モジュール集合が矛盾
33:   IPEnd:
34: end while
```

図 3.1 Hyrose による HydLa プログラムの非決定実行アルゴリズム

そこで，まずはこれら規則のうち 3 に関して，その解の候補となるような制約モジュール集合（解候補モジュール集合）の集合を求める．これは，図 3.1 における SolveCH 関数の部分に対応する．SolveCH の具体的なアルゴリズムは [12] に詳しい．なお，こうして得られた解候補モジュール集合の集合 *MS* は，トポロジカルソートがおこなわれているものとする．

3.3 フェーズ切り替え処理

制約階層処理が終わると，続いてフェーズ切り替え処理をおこなう．Hyrose においては，ハイブリッドシステムのシミュレーションをおこなう際，離散変化を計算するフェーズである Point Phase（PP）と，連続変化を計算するフェーズである Interval Phase（IP）の 2 つのフェーズに基づいておこなわれる．シミュレーションは時刻 T=0 の PP から開始し，その後，あらかじめ指定されたシミュレーション打ちきり時刻 *MaxT* まで IP，PP，IP,... と繰り返すことによって進められる．各フェーズにおいては，3.2 において示した 3 つの規則のうち，1 および 2 を満たすような制約モジュール集合を求めながら，そのフェーズにおいて各変数が起こす変化を計算する．

3.3.1 無矛盾極大モジュール集合の導出

制約階層処理の結果得られた *MS* の先頭要素であるモジュール集合は，HydLa プログラムに出現したすべての制約モジュールを含んだものとなっているはずである．このモジュール集合を採用した結果，矛盾なくそのフェーズにおける各変数の解が得られる場合，まさしくこのモジュール集合こそが無矛盾極大モジュール集合であると見なすことができ，そのフェーズにおけるシミュレーション結果は 1 通りとなる．

一方，先頭要素のモジュール集合を採用してシミュレーションをおこなった結果，矛盾が生じた場合については，*MS* の 2 番目以降の要素を順次採用することで，シミュレーションを再試行する．*MS* の 2 番目以降の要素は先頭要素よりもそのモジュール数が少なくなっており，これはつまり，採用する制約を減らしている（満たすべき条件を緩めている）ことに他ならない．

図 3.1 に示したアルゴリズムは，入力の HydLa プログラムが複数の解軌道を持つ際に，その 1 つだけを得る非決定実行アルゴリズムである．2.2.6 節で述べた全解探索実行においては，すべての解軌道を求める必要があるため，GetElement をおこなう際に 1 つの要素にだけではなく全ての要素に関してシミュレーションを続行する点が異なる．

Require: 制約ストア S，制約モジュール集合 M，解候補モジュール集合のリスト MS，
現在のシミュレーション時刻 T
Ensure: 制約ストアの集合 SS，新しい解候補モジュール集合のリスト MS
1: **if** $T > 0$ **then**
2: M :=EliminateNotAlways(M)
3: **end if**
4: (SS, _, _, MS) :=CalculateClosure(S, M, MS,CheckConsistencyPP)
5: **return** (SS, MS)

図 3.2 PP のアルゴリズム

3.3.2 PP の処理

PP では離散変化の計算をおこなう．その実行アルゴリズムを図 3.2 に示す．HydLa プログラムのうち，[] がついていないような制約は，シミュレーション時刻 T=0 でのみ評価し，それ以降のシミュレーションにおいては特に意味をなさない．よって，シミュレーション時刻 T が 0 より大きいときはこのような制約を取り除くことができる．その後，成り立っていない新しいガード条件がある限り，無矛盾性判定とガード条件判定を繰り返す．この処理は閉包計算と呼ばれ，図 3.2 における CalculateClosure に対応する．CalculateClosure のアルゴリズムを図 3.3 に示す．

PP における閉包計算が終わった時点で，次の IP に入るための準備をおこなう．PP 内のこの時点における HydLa プログラム内の各変数の値を取り出し，それを PP 実行結果として保持するとともに，次の IP の初期値として使えるようにする．変数が一意に定まる値で表される場合は，その値が変数表（Variable Map）という表に登録される．一方，変数が不確実値を持つ場合，たとえば不等式制約によって値に範囲を持つような場合は，その変数に対応した名前の，新しい定数を自動的に追加する．これはパラメタ定数と呼ばれ，変数表には変数値がこの定数値であることが登録される．また，変数表とは別に定数表（Parameter Map）という表に対して，パラメタ定数自体の条件が登録される．このようにして，変数が不確実な値を持つことを表す．以降のシミュレーションにおいて，数式中にパラメタ定数が含まれることになるが，その際，常にその条件（パラメタ定数が取りうる値の範囲）に注意して扱う必要がある．

Require: 制約ストア S，現在の制約モジュール集合 M，解候補モジュール集合のリスト MS，無矛盾性判定関数 CheckConsistency(S)

Ensure: 制約ストアの集合 SS，成立しない条件付き制約の集合 A_-，成立する条件付き制約の集合 A_+，新しい解候補モジュール集合のリスト MS

1: $A_- := \emptyset$
2: $A_+ := \emptyset$
3: **repeat**
4: $S :=$ CollectTell(M, S)
5: **if** $\neg$CheckConsistency(S) **then**
6: **return** $(\emptyset, \emptyset, \emptyset, MS)$
7: **end if**
8: $A_- := A_- \cup$ CollectAsk(M)
9: $Expanded := false$
10: **for** $(g \Rightarrow c) \in A_-$ **do**
11: $(S_{true}, S_{false}) := ((S \wedge g), (S \wedge \neg g))$
12: **if** CheckConsistency(S_{true}) $\wedge$ CheckConsistency(S_{false}) **then**
13: **return** $(\{S_{true}, S_{false}\}, A_-, A_+, MS)$
14: **end if**
15: **if** CheckConsistency(S_{true}) **then**
16: $M :=$ DeleteGuard($M, (g \Rightarrow c)$)
17: **if** CheckAlways(c) **then**
18: $MS := \{\text{DeleteGuard}(M_i, (g \Rightarrow c)) | M_i \in MS\}$
19: **end if**
20: $A_- := A_- \setminus \{g \Rightarrow c\}$
21: $A_+ := A_+ \cup \{g \Rightarrow c\}$
22: $Expanded := true$
23: **end if**
24: **end for**
25: **until** $\neg Expanded$
26: **return** $(\{S\}, A_-, A_+, MS)$

図 3.3 CalculateClosure のアルゴリズム

Require: 制約ストア S，現在の制約モジュール集合 M，解候補モジュール集合のリスト MS，現在のシミュレーション時刻 T，最大シミュレーション時刻 $MaxT$

Ensure: 制約ストアの集合 SS，IP 終了時刻 $EndT$，新しい解候補モジュール集合のリスト MS

1: $M :=$ EliminateNotAlways(M)
2: $M_{all} :=$ MaxModule(MS)
3: $(SS, A_-, A_+, MS) :=$ CalculateClosure(S, M, MS,CheckConsistencyIP)
4: **if** $|SS| \neq 1$ **then**
5: **return** $(SS, MaxT, MS)$
6: **end if**
7: $S_t :=$ SolveDifferentialEquation(GetElement(SS))
8: $textitTS$:= CalculateNextPPTime(
$\{(S_t \wedge g) | (g \Rightarrow c) \in A_-\}$
$\cup \{(S_t \wedge \neg g) | (g \Rightarrow c) \in A_+\}$
$\cup \{(S_t \wedge M_-) | M_- \in (M_{all} \setminus M)\}$
$\cup \{(S_t \wedge \neg M_+) | M_+ \in M\}$,
GetParameterCons(S_t), $MaxT - T$)
9: $(MinT, S_p)$:= GetElement(TS)
10: $S_t := (S_p \wedge S_t)$
11: $S :=$ SubstituteMinTime($S_t, MinT$)
12: **return** $(\{S\}, MinT + T, MS)$

図 3.4 IP のアルゴリズム

3.3.3 IP の処理

IP では連続変化の計算をおこなう．その実行アルゴリズムを図 3.4 に示す．IP では，まず PP と同様に閉包計算 CalculateClosure をおこなう．その後，次の離散変化時刻を CalculateNextPPTime により求める．これらの処理には，どちらも微分方程式を解く必要がある．

3.4 計算処理

フェーズ切り替え処理をおこなうにあたって，閉包計算における無矛盾性判定 CheckConsistency や IP における離散変化時刻求解 CalculateNextPPTime などにおいては，数学的な計算をおこなう必要がある．これらをおこなうのは外部のソルバ（今回のアルゴリズムは数式処理に基づいたシミュレーションであるので数式処理システム）であり，Hyrose 側から計算に必要な制約等をソルバに送り，計算を依頼する．そして，ソルバが計算をおこなった結果を受け取り，その内容に応じた処理をおこなうモジュールが必要となる．このようなモジュールとして，現在 Hyrose では Mathematica に基づいて計算をおこなう Mathematica Constraint Solver（MCS）と，REDUCE に基づいて計算をおこなう REDUCE Constraint Solver（RCS）とが存在する．本研究では，後者の RCS について，その制約処理手法を解説する．

第 4 章

REDUCE Constraint Solver の制約処理手法

本章では，本研究において実装をおこなった，Hyrose の数式処理制約求解系である REDUCE Constraint Solver（RCS）における制約処理手法について，解説をおこなう．

4.1 RCS の概要

RCS は，数式処理システム REDUCE をソルバとした制約求解系である．Hyrose が HydLa モデルのシミュレーションをおこなうにあたって必要な種々の計算処理を，REDUCE を用いることでおこなう．

REDUCE は数式処理システムであるため，数式の変換や代入といった操作に基づいて計算をおこなう．そのため，計算機における数値計算につきものである計算誤差を含むことがなく，正しくない結果を返すことのない，高信頼な計算が可能である．数式処理のこの特性により，Hyrose の目標であるハイブリッドシステムの検証の結果を保証することができる．

RCS は大きく分けて 2 つのモジュールにより構成されている．まず 1 つ目は，関数定義部である．関数定義部は，Hyrose によるシミュレーションに必要な，具体的な計算をおこなうための数々の関数から成るモジュールであり，REDUCE により実装されている．その詳しい実装については 6 において述べる．

2 つ目は，アダプタ部である．アダプタ部は，Hyrose のシミュレーション処理中の計算が必要になったタイミングにおいて，前述の関数定義部に対して具体的な引数を与えるための関数から成るモジュールであり，C++ により実装されている．そのすべ

ての処理において，数式処理システム REDUCE との通信が必要となるが，この機能は REDUCE Link[10] により提供されている．REDUCE Link を介して，制約やシミュレーションの打ち切り時刻等といった，計算に必要な諸データを REDUCE に渡したり，REDUCE によって計算された結果の式を受け取ったりすることが可能になっている．なお，REDUCELink では REDUCE との通信を主に文字列形式によっておこなっており，REDUCE からの出力は S 式形式によって返るため，S 式向けのパース処理が実装されている．これにより，REDUCE による計算結果はツリー形式に変換されるため，Hyrose 本体側での扱いがより容易なものになっている．

以降，Hyrose のシミュレーションにおいて，3.3.2 節や 3.3.3 節で述べた計算処理を，数式処理システム REDUCE に基づいておこなうための制約処理手法について解説をおこなう．

4.2 CheckConsistency

CheckConsistency では，制約ストア S に関して，その無矛盾性を評価する．CheckConsistency には，PP で使うものと IP で使うものとが存在し，それぞれ CheckConsistencyPP，CheckConsistencyIP となっている．CheckConsistencyPP のアルゴリズムを図 4.1 に，CheckConsistencyIP のアルゴリズムを図 4.2 に，それぞれ示す．

Require: 制約ストア S
Ensure: 無矛盾であるかどうか $bConsistent$
1: **return** $\exists(S)$

図 4.1 CheckConsistencyPP のアルゴリズム

Require: 制約ストア S
Ensure: 無矛盾であるかどうか $bConsistent$
1: $S_t :=$ SolveDifferentialEquation(S)
2: **return** $(\mathrm{Inf}\{t \mid \exists_{\setminus t}(S_t \wedge (t > 0))\} = 0)$

図 4.2 CheckConsistencyIP のアルゴリズム

4.3 CalculateNextPPtime

CalculateNextPPTime では, 離散変化の要因となる条件のリスト TL を元に, それが離散変化の要因となる場合の時刻を FindMinTime によりそれぞれ求め, 最小の時刻となるものを CompareMinTimeList によって見つけ出す．CalculateNextPPTime のアルゴリズムを，図 4.3 に示す.

Require: 離散変化条件のリスト TL，記号定数についての制約 C，シミュレーション終了時刻 $MaxT$
Ensure: 時刻と条件の組のリスト TCL

1: f:=Map($(fun\ tr \rightarrow$FindMinTime$(tr, C)), TL$)
2: **return** FoldR(CompareMinTimeList, $f, \{\{MaxT, C\}\}$)

図 4.3 CalculateNextPPTime のアルゴリズム

4.3.1 FindMinTime

FindMinTime では，ある 1 つの離散変化条件 $Trigger$ に関して，それが離散変化の要因となる場合の時刻を求める．FindMinTime のアルゴリズムを図 4.4 に示す.

Require: 離散変化条件 $Trigger$，記号定数についての制約 $Constraint$,
Ensure: 時刻と条件の組のリスト

1: **return** GetInf($Trigger \wedge Constraint \wedge (t > 0)$)

図 4.4 FindMinTime のアルゴリズム

4.3.2 CompareMinTimeList

CompareMinTimeList では，4.3 節における TL の全要素に関して求まった，離散変化時刻とそれを与えるパラメタ定数の条件の組のリスト間での比較をおこない，離散変化時刻が最小となっているようなものを抽出する．CompareMinTimeList のアルゴリズムを図 4.5 に示す.

また，CompareMinTimeList 内で使用している CompareMinTime のアルゴリズムを図 4.6 に示す.

Require: 時刻と条件の組のリスト *L1*, *L2*

Ensure: 時刻と条件の組のリスト

1: **return** FoldR($(fun\ l\ r \rightarrow$ Map$(fun\ e \rightarrow$ CompareMinTime$(l, e), L2) \cup r), L1, \{\})$

図 4.5 CompareMinTimeList のアルゴリズム

Require: 時刻と条件の組 *TC1*, 時刻と条件の組 *TC2*

Ensure: 時刻と条件の組のリスト

1: $c := (TC1.cond \wedge TC2.cond)$

2: $c1 := (c \wedge (TC1.time < TC2.time))$

3: $c2 := (c \wedge \neg c1)$

4: **return** filter$((fun\{_, c\} \rightarrow c \neq false), \{\{TC1.time, c1\}, \{TC2.time, c2\}\}$

図 4.6 CompareMinTime のアルゴリズム

第 5 章

不確実値を含む数式の大小判定法

本章では REDUCE Constraint Solver（RCS）に導入した，無理数や記号定数といった不確実値を含む数式に関して大小判定をおこなうための手法について解説する．

5.1 無理数を含む数式の大小判定法

まず，無理数を含む数式に対しては，その値を挟むような区間形式に変換し，区間演算 [4] を用いた符号判定をおこなうことで，大小判定をおこなうことを考える．

5.1.1 区間値への変換を用いた符号判定法導入の目的

3.3.3 節で述べたように，Hyrose ではシミュレーションをおこなう際に，ある連続変化が継続している状態から何らかの離散変化が起きる時刻を計算処理によって求めることでシミュレーション時刻を先に進めており，その際に 2 値の厳密な大小判定が必要となる．

しかし，数式処理システム REDUCE による計算処理においては，無理数に関する厳密な大小比較をおこなうことができないという問題を有している．そこで，無理数を初めとする，数式処理を用いた大小比較のおこなえない 2 つの定数を対象として，間違った答えが返ることのない，厳密な大小判定を実現することを考える．

5.1.2 問題設定

本手法において解く対象として考えるのは，平方根などの無理数を含む 2 つの定数式である．その例として以下のような 2 つの定数式を考える．

$$\sqrt{2}+\sqrt{5}-\sqrt{11} \tag{5.1}$$

$$\sqrt{3}+\sqrt{7}-\sqrt{13} \tag{5.2}$$

このような 2 つの定数式が与えられているとき，2 つの値の間での厳密な大小判定をおこなうには，2 値の差を取り，新たに得られた定数式に関して，符号判定をおこなえば良い．つまり，5.1 − 5.2 より，

$$\sqrt{2}+\sqrt{5}-\sqrt{11}\ \ -\sqrt{3}+\sqrt{7}-\sqrt{13} \tag{5.3}$$

5.3 のような定数式を得ることができる．ここで，この定数式が 0 より大きいか小さいか，つまり符号判定処理をおこなうことで，実質 5.1 と 5.2 との間の大小判定がおこなえることに着目した．以降では，2 つの定数式を区間形式で表された 1 つの定数式に変換し，その符号判定をおこなう手法について考える．

5.1.3 区間値への変換処理

数式処理システム REDUCE においては，無理数を数値として扱えないが，そのかわりに無限精度の整数を扱うことができるという特徴がある．この特徴を利用し，無理数を含む数式を，無限長整数によって表された有理数を上下限とする区間値に変換することを考える．つまり，計算機では表すことのできない無理数の値を，その値にごく近い有理数で挟むことで表現する．

Hyrose によるシミュレーションにおいて扱う必要のある無理数には様々あるが，まずは 5.1.2 節でも扱ったような平方根の区間値への変換を考える．平方根を区間値へ変換する手法としては，二分探索法やニュートン法が存在するが，今回の実装においてはニュートン法を採用した．ニュートン法による平方根の区間値への変換処理 GetSqrtInterval を図 5.1 に示す．ここで，繰り返しの終了条件に使用されている $intPrec$ は，区間値への変換における精度を表しており，Hyrose のユーザがあらかじめ指定可能となっているものである．

こうして，定数式中の各平方根が区間形式により得られたら，定数式の各項に関して区間演算を施すことで，定数式全体が表す数値を区間形式により求めることができる．

なお，平方根以外の無理数に関しては，以下のように区間値への変換をおこなうこととした．

- 三角関数 — $[-1, 1]$ とする
- 逆三角関数 — $[-\pi, \pi]$ とする

Require: 平方根を求めたい数値 val
Ensure: 入力値の平方根を表す区間値 $sqrtValInt$

1: $newTmpSqrtValInt := val$
2: **repeat**
3: $tmpSqrtValInt := newTmpSqrtValInt$
4: $newTmpSqrtValInt := \frac{1}{2} \times (tmpSqrtValInt + \frac{val}{tmpSqrtValInt})$
5: **until** $tmpSqrtValInt - newTmpSqrtValInt < \frac{1}{10^{intPrec}}$
6: $sqrtLb := 2 \times newTmpSqrtValInt - tmpSqrtValInt$
7: $sqrtUb := newTmpSqrtValInt$
8: **return** $(sqrtLb, sqrtUb)$

図 5.1 GetSqrtInterval のアルゴリズム

5.1.4 区間値の符号判定による定数式の大小判定

定数式全体を表す区間値が求まったら，次にその区間の符号判定をおこなう．符号判定は，区間の上限および下限の符号を調べることでおこなう．表 5.1 に，区間の上限および下限の符号と，その区間により表される定数式の元となった 2 つの定数式に関しての大小判定の結果との関係を示す．表からも分かるように，上限および下限の符号が一致してい

表 5.1 定数式の上下限の符号と判定結果との関係

上限の符号	下限の符号	大小判定結果
+	+	>
+	−	?
−	+	?
−	−	<

れば，その区間値の符号，つまり区間値が 0 より大きいかどうかは明らかに分かり，それにより元の 2 つの定数式間の大小判定がおこなえる．上下限の符号がともに正であれば元の 2 つの定数式のうち引かれた方の数が，上下限の符号がともに負であれば元の 2 つの定数式のうち引いた方の数が大きいということになる．一方，上下限の符号が一致していない場合は，「0 より大きいかどうか」という問題に対して，確実な答えを出すことはできない．そこで，このような場合には Unknown，つまり判定をしようとしたが結果は不明で

ある，という答えを返すことにする．こうすることで，間違った判定結果を返してしまう心配はなくなる．現在の実装では，大小判定の結果 Unknown が返った場合は，区間値への変換の精度を上げた上で，再び 5.1.3 節からの処理をやり直すようにしている．

5.2 記号定数を含む数式の大小判定法

次に，数式が記号定数を含むような場合に，記号定数に関しての 2 段階の無矛盾性判定の結果を調べることで大小判定をおこなうことを考える．

5.2.1 2 段階の無矛盾性判定による大小判定法導入の目的

一部の HydLa モデルにおいては，初期値が幅を持つなど不確実値を含むことがあり，このような場合において Hyrose は，3.3.2 節で述べた通り，記号定数を自動的に追加する．そのような場合，数式内に記号定数が入った状態でシミュレーションは続行することになる．ここで，記号定数を含む数式の大小判定を考えると，あらかじめ与えられている記号定数の条件（値の範囲）のうち，どの値を持たせるかによって，数式全体の大きさが変わってしまい，その大小判定結果も異なってきてしまうことが分かる．たとえば，記号定数のある値を境に 2 通りの判定結果が得られたり，逆に，記号定数に関する条件中のどの値をとっても判定結果が変わることはないようなことも考えられる．このように，大小判定結果が記号定数の範囲によって変化しうるような場合に，間違った答えが返ることなく，厳密な大小判定を実現することを考える．

5.2.2 CompareParamVal

記号定数を含む式 *PExpr* と，値 *val* との大小判定をおこなうことを考える．ただしここで，*PExpr* 中に含まれている記号定数の条件は *PCond* で表されているとする．この処理 CompareParamVal のアルゴリズムを図 5.2 に示す．

Require: 記号定数を含む式 $PExpr$，大小判定の対象値 val，記号定数についての制約 $PCond$,

Ensure: (対象値よりも小さくなるための条件，対象値よりも大きくなるための条件)

1: $LessCond := (PExpr < val) \wedge PCond$
2: **if** $LessCond = PCond$ **then**
3: **return** $(LessCond, \emptyset)$
4: **end if**
5: $GeqCond := \neg LessCond \wedge PCond$
6: **return** $(LessCond, GeqCond)$

図 5.2 CompareParamVal のアルゴリズム

第 6 章

RCS 関数定義部の実装

RCS によるシミュレーションにおけるあらゆる時点において必要となる演算は，あらかじめ関数定義として用意しており，Hyrose 実行時にこれを REDUCE に読み込ませることで計算を進める．本章では REDUCE Constraint Solver（RCS）のうち，REDUCE で実装された関数定義部の詳細について述べる．

6.1 新たに導入したデータ形式

関数定義部内では数多くの数学的演算や論理演算が必要となるが，それにあたって元々用意されている REDUCE のライブラリ関数による処理では不都合がある，あるいは正しい結果が得られないことがあることが確認できた．そこで，そのような処理を正しくおこなうべく，まずは扱う対象であるデータの形式から，新しく再定義をおこなった．

6.1.1 不等式タプル

数式処理システム REDUCE において，不等式を扱うことは当然可能であるが，計算過程において，その不等号の向きが限定されているという問題がある．つまり，$<$ や $\leq$ を含む式は $>$ や $\geq$ によって表された等価な式へと強制的に変換されてしまう．そこで，(変数, 不等号, 値) の 3 つ組のデータ形式を新たに定義し，不等式を扱う処理においてはこのデータ形式に基づいて取り扱うようにした．なお，変数の次数は 1 次とし，不等号は $\leq$, $\geq$, $<$, $>$ を扱うものとする．これにより，不等式がより扱いやすいものになった．

6.1.2 不等式タプルを用いた DNF

数式処理システム REDUCE において, 論理演算をおこなうパッケージとして Redlog[2] が存在する．しかし，Redlog を用いて連立不等式を表すような論理式を解こうとした際, その無矛盾性，つまり矛盾があるかどうか，に関しての判定しかおこなうことができない．これは，無矛盾である場合の変数の解を求めることができないことを意味する．4.2 節で述べた無矛盾性判定をおこなう分にはそれでも問題ないが，4.3 節で述べた離散変化時刻求解をおこなう際には，具体的な離散変化時刻（値または範囲）を求める必要があるため，問題になってしまう．そこで新たに，6.1.1 節で述べた不等式タプルを用いて論理式を表すデータ形式を新たに定義した．このデータ形式では，論理式を DNF 形式で統一して扱っており，前述の不等式タプルを 2 重リスト（階層の深さ 1 が論理和，深さ 2 が論理積をあらわしている）で囲むことで実現している．なお，同じ変数に関して，$\leq$ を持つタプルと $\geq$ を持つタプルとの論理積を用いることによって，等式を表すことも可能になっている．

6.2 基本的演算処理の関数定義

本節では数ある関数定義の中でも頻繁に使用する特に重要なものに関して，その処理手法の解説をおこなう．その多くは数学の基本的演算や論理演算となっている．

6.2.1 exSolve

exSolve は連立方程式の求解をおこなう関数である．REDUCE に組み込みの solve 関数を用いることで，連立方程式の求解はおこなえるが，入力された方程式によってはその解に，実数以外の値が含まれる場合がある．しかし，Hyrose のシミュレーションにおいて実数以外の解は扱わないため，これらを取り除く必要がある．exSolve においては，虚数単位 i を含む解や，$\sin 5$ といったような，実数上で定義できない値を含む解がないかを調べることで，実数解のみを返すことが可能になっている．また，得られた実数解は最後に必ず有理化をおこなうようにしている．これは，論理演算パッケージである Redlog を用いた計算において，分数の分母部分に無理数が入るような数式，つまり有理化が施されていない数式を引数として受け付けない場合があることが確認されており，この問題を回避するためである．

6.2.2 exIneqSolve

exIneqSolve は不等式の求解をおこなう関数である．なお，入力として扱えるのは 1 つの変数を持つ 1 つの不等式であり，その次数は 2 次までとなっている．基本的な考え方としては，まず与えられた不等式が等式であると見なして exSolve で解き，得られた解の値に対して，与えられた不等式の次数および不等号の向きを元に，不等式の解となる範囲を 6.1.2 節で述べた不等式タプルによる DNF 形式で返す，といった処理の流れになる．なお，この方法だけでは，根号の中に変数を含むような不等式の場合に正しい解の範囲を得ることができないため，与えられた不等式中に根号が出現する際はその中身の式が 0 以上であるという条件を最後に連立することで，補っている．

6.2.3 exDSolve

exDSolve は微分方程式の求解をおこなう関数である．HydLa のシミュレーションにおいて，プログラム中に複数の微分方程式が含まれる場合も考えられる．その際，連立微分方程式の初期値問題を解く必要がある．REDUCE に組み込みの odesolve 関数では 1 つの微分方程式しか扱えないため，そのような場合に対応できなかったが，exDSolve ではラプラス変換を用いた求解をおこなっており，これにより連立微分方程式に対応することが可能となっている．

6.2.4 不等式タプルによる DNF に基づく論理積求解

6.1.2 節で述べた，不等式タプルによる DNF 形式を用いての論理演算も新たに定義をおこなった．既存の DNF 形式の論理式に対して，新たな論理式との論理積を取った結果を返す．論理積の計算結果は，不等式タプルによって，変数の取りうる値の範囲を表す上下限（両方または片方だけ）が厳密に求まった形式で得られる．つまり，1 元 1 次の連立不等式の簡約化をおこなえることになる．なおこの際に，変数の上下限は無理数を含む場合であっても厳密に大小判定がおこなわれ，また，等号を含む不等号（$\leq$，$\geq$）と含まない不等号（$<$，$>$）とを区別して正しく扱うこともできるようになっている．これにより，1 変数に関しては，不等式および等式によって構成される論理式について，簡約化つきの論理演算を正しく扱える体系が構築できた．

第 7 章

例題による考察

本章では例題を元に，RCS によって正しいシミュレーション実行がおこなわれる様子を確認する．その際，計算処理をおこなうモジュールとして，3.4 節で述べた MCS を選択して実行した場合の結果と，RCS を選択して実行した場合の結果とを比較することで評価をおこなった．例題としては，計算過程において平方根や三角関数といった無理数の入った数式が出現するようなモデルを選んだ．また，初期値に不確実値を含むことにより計算過程においてパラメタ定数が出現するようなモデルに関しても確認を行った．

7.1 bouncing_particle

まず 1 つ目の例題は，2.2.3 節で示した，空中から落下した質点が地面でバウンドするモデルである．このモデルを HydLa で記述したプログラムを以下に示す（再掲）．

```
INIT   <=> y=10 /\ y'=0.
FALL   <=> [](y'' = -10).
BOUNCE <=> [](y- = 0 => y' = -(4/5) * y'-).

INIT, FALL << BOUNCE.
```

前述の通り，変数 y は質点の高さ，y' は y 方向に関する速さを表している．このモデルにおいて，質点は初期位置（高さ）10 から重力加速度 10（下向き）によって自然落下を始める．よって，時刻 t の変化に対するこの質点の高さの式は $y(t) = -5 * t^2 + 10$ となる．よって，地面（$y = 0$）に衝突するシミュレーション時刻 T は $T = \sqrt{2}$ となる．このシミュレーションをおこなうにあたり，4.3.2 節で述べた CompareMinTime によりシ

ミュレーション終了時刻 $MaxTime$ との大小比較をおこなう必要があるが，このとき求まった離散変化時刻 $T = \sqrt{2}$ と $MaxTime$ との比較が正しくおこなわれているかどうかを確認する．今回はシミュレーション終了時刻を $maxTime = 6$ として実行結果の比較をおこなった．

まず，MCS を選択した場合の実行結果を図 7.1 に示す．

続いて，RCS を選択した場合の実行結果を図 7.2 に示す．

MCS と RCS との二者のシミュレーション結果を比較すると，最後の IP における各変数の式は形式に多少の差異は認められるものの，一致していることが分かる．

また，実行の結果として得られた t の式をプロットしたものを図 7.3 に示す．

図より，質点が地面をすり抜けるなどすることなく，高さ $y = 0$ においてバウンドする解軌道を正しく求められていることが確認できる．これは，前述の通り，無理数を含む数式の大小比較が正しくおこなってこそ得られるものであり，第 5 章における区間値への変換等の処理一環が正しくおこなわれたことを意味する．

7.2 bouncing_particle_interval

2 つめの例題は，2.2.6 節において示した，初期値に幅を持った質点が自由落下し，地面でバウンドするモデルである．

このモデルにおいては，質点は初期位置（高さ）y が 9 以上 11 以下となっている．そのため，Hyrose により y のパラメタ定数である py が追加され，パラメタ定数に関する制約 $9 \leq py \leq 11$ が新たに追加される．この時，時刻 t の変化に対するこの質点の高さの式は $y(t) = -5 * t^2 + py$ となり，パラメタ定数を含んだ形となる．このとき，地面（$y = 0$）に衝突するシミュレーション時刻 T は $T = \sqrt{\frac{py}{5}}$ となる．ここで，この求まった時刻とシミュレーション終了時刻 $MaxTime$ との大小比較をおこなうことを考えると，$9 \leq py \leq 11$ の条件のもとで，$\sqrt{\frac{py}{5}} < MaxTime$ が成り立つかどうかを調べる必要がある．パラメタ定数を含んだ離散変化時刻が求まった場合の処理が正しくおこなわれているかどうかを確認する．今回はシミュレーション終了時刻を $maxTime = 6$ として実行結果の比較をおこなった．

まず，MCS を選択した場合の実行結果を図 7.4 に示す．

続いて，RCS を選択した場合の実行結果を図 7.5 に示す．

MCS と RCS との二者のシミュレーション結果を比較すると，最後の IP における各変数の式は形式に多少の差異は認められるものの，一致していることが分かる．

```
#---------1---------
---------PP---------
time    : 0
y       : 10
y'      : 0
y''     : -10

---------IP---------
time    : 0 -> 2^(1/2)
y       : (-5)*(-2+t^2)
y'      : (-10)*t
y''     : -10

#---------2---------
---------PP---------
time    : 2^(1/2)
y       : 0
y'      : 8*2^(1/2)
y''     : UNDEF

---------IP---------
time    : 2^(1/2) -> 13/5*2^(1/2)
y       : -26+18*2^(1/2)*t+(-5)*t^2
y'      : 2*(9*2^(1/2)+(-5)*t)
y''     : -10

#---------3---------
---------PP---------
time    : 13/5*2^(1/2)
y       : 0
y'      : 32/5*2^(1/2)
y''     : UNDEF

---------IP---------
time    : 13/5*2^(1/2) -> 97/25*2^(1/2)
y       : (-2522)/25+162/5*2^(1/2)*t+(-5)*t^2
y'      : 162/5*2^(1/2)+(-10)*t
y''     : -10

#---------4---------
---------PP---------
time    : 97/25*2^(1/2)
y       : 0
y'      : 128/25*2^(1/2)
y''     : UNDEF

---------IP---------
time    : 97/25*2^(1/2) -> 6
y       : (-118922)/625+1098/25*2^(1/2)*t+(-5)*t^2
y'      : 1098/25*2^(1/2)+(-10)*t
y''     : -10


#time ended
```

図 7.1 bouncing_particle の MCS による実行結果

```
#---------1---------
---------PP---------
time    : 0
y       : 10
y'      : 0
y''     : -10

---------IP---------
time    : 0 -> 2^(1/2)
y       : -5*t^2+10
y'      : -10*t
y''     : -10

#---------2---------
---------PP---------
time    : 2^(1/2)
y       : 0
y'      : 8*2^(1/2)
y''     : UNDEF

---------IP---------
time    : 2^(1/2) -> 13*2^(1/2)/5
y       : 18*2^(1/2)*t+(-(5*t^2))+(-26)
y'      : 18*2^(1/2)+(-(10*t))
y''     : -10

#---------3---------
---------PP---------
time    : 13*2^(1/2)/5
y       : 0
y'      : 32*2^(1/2)/5
y''     : UNDEF

---------IP---------
time    : 13*2^(1/2)/5 -> 97*2^(1/2)/25
y       : (810*2^(1/2)*t+(-(125*t^2))+(-2522))/25
y'      : (162*2^(1/2)+(-(50*t)))/5
y''     : -10

#---------4---------
---------PP---------
time    : 97*2^(1/2)/25
y       : 0
y'      : 128*2^(1/2)/25
y''     : UNDEF

---------IP---------
time    : 97*2^(1/2)/25 -> 6
y       : (27450*2^(1/2)*t+(-(3125*t^2))+(-118922))/625
y'      : (1098*2^(1/2)+(-(250*t)))/25
y''     : -10


#time ended
```

図 7.2 bouncing_particle の RCS による実行結果

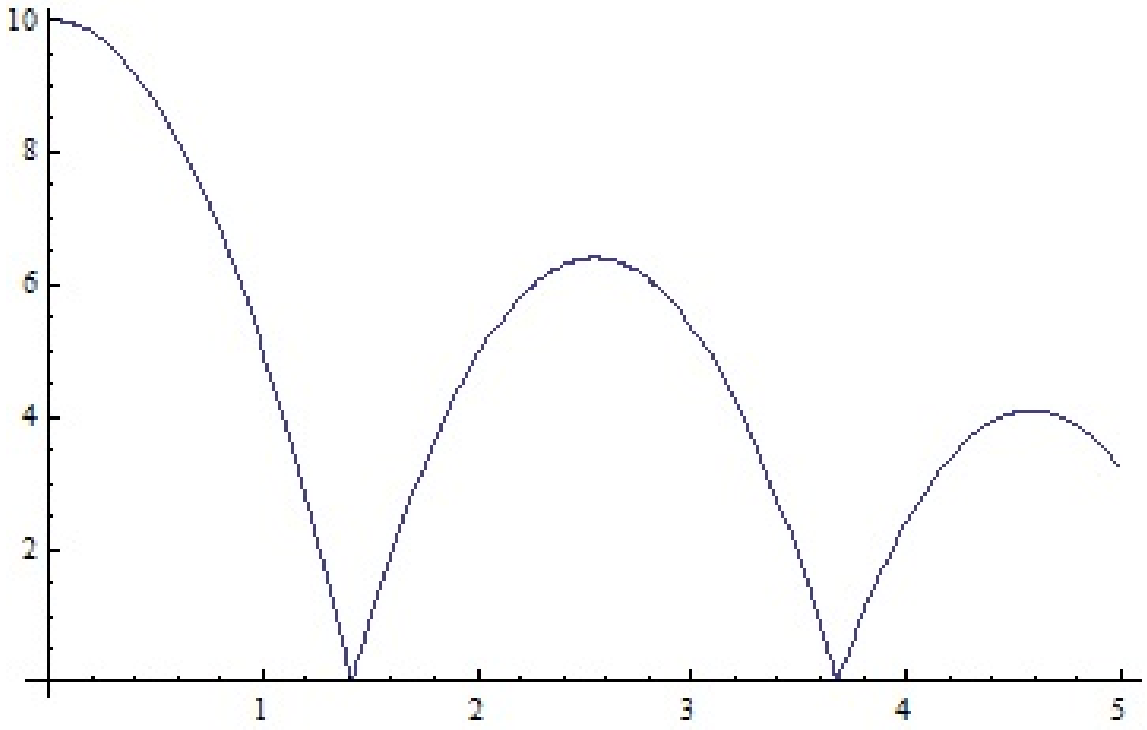

図 7.3 bouncing_particle を RCS によってシミュレーションした結果のグラフ

7.3 braking

3 つめの例題は，進行していた車が途中でブレーキをかけるモデルである．そのプログラムを図 7.6 に示す.

このモデルにおいては，車は位置 x の初期値 0 と x 方向の速さ x' の初期値 1 を持ち，最初は加速度 1 で加速していく．しかし，x が 3 に到達すると車はブレーキをかけ (BRAKE 制約の部分)，車は減速していく．このモデルに関して，今回はシミュレーション終了時刻を $maxTime = 4$ として実行結果の比較をおこなった.

まず，MCS を選択した場合の実行結果を図 7.7 に示す.

続いて，RCS を選択した場合の実行結果を図 7.8 に示す.

RCS を選択した場合の実行では，2 回目の PP までは正しくシミュレーションが進んでいることが確認できるが，2 回目の IP において，無限ループに陥ってしまい，シミュレーション処理が終わらないという現象が起きてしまう．これは，2 回目の IP での計算

```
#---------1---------
---------PP---------
time    : 0
y       : py
y'      : 0
y''     : -10

---------IP---------
time    : 0 -> 5^((-1)/2)*py^(1/2)
y       : py+(-5)*t^2
y'      : (-10)*t
y''     : -10

#---------2---------
---------PP---------
time    : 5^((-1)/2)*py^(1/2)
y       : 0
y'      : 8*5^((-1)/2)*py^(1/2)
y''     : UNDEF

---------IP---------
time    : 5^((-1)/2)*py^(1/2) -> 13/5*5^((-1)/2)*py^(1/2)
y       : (-13)/5*py+18*5^((-1)/2)*py^(1/2)*t+(-5)*t^2
y'      : 18*5^((-1)/2)*py^(1/2)+(-10)*t
y''     : -10

#---------3---------
---------PP---------
time    : 13/5*5^((-1)/2)*py^(1/2)
y       : 0
y'      : 32/5*5^((-1)/2)*py^(1/2)
y''     : UNDEF

---------IP---------
time    : 13/5*5^((-1)/2)*py^(1/2) -> 97/25*5^((-1)/2)*py^(1/2)
y       : (-1261)/125*py+162/5*5^((-1)/2)*py^(1/2)*t+(-5)*t^2
y'      : 162/5*5^((-1)/2)*py^(1/2)+(-10)*t
y''     : -10

#---------4---------
---------PP---------
time    : 97/25*5^((-1)/2)*py^(1/2)
y       : 0
y'      : 128/25*5^((-1)/2)*py^(1/2)
y''     : UNDEF

---------IP---------
time    : 97/25*5^((-1)/2)*py^(1/2) -> 6
y       : (-59461)/3125*py+1098/25*5^((-1)/2)*py^(1/2)*t+(-5)*t^2
y'      : 1098/25*5^((-1)/2)*py^(1/2)+(-10)*t
y''     : -10


#---------parameter condition---------
py      : [9, 11]

#time ended
```

図 7.4 bouncing_particle_interval の MCS による実行結果

```
#---------1---------
---------PP---------
time    : 0
y       : py
y'      : 0
y''     : -10

---------IP---------
time    : 0 -> py^(1/2)*5^(1/2)/5
y       : py+(-(5*t^2))
y'      : -10*t
y''     : -10

#---------2---------
---------PP---------
time    : py^(1/2)*5^(1/2)/5
y       : 0
y'      : 8*py^(1/2)*5^(1/2)/5
y''     : UNDEF

---------IP---------
time    : py^(1/2)*5^(1/2)/5 -> 13*py^(1/2)*5^(1/2)/25
y       : (18*py^(1/2)*5^(1/2)*t+(-(13*py))+(-(25*t^2)))/5
y'      : (18*py^(1/2)*5^(1/2)+(-(50*t)))/5
y''     : -10

#---------3---------
---------PP---------
time    : 13*py^(1/2)*5^(1/2)/25
y       : 0
y'      : 32*py^(1/2)*5^(1/2)/25
y''     : UNDEF

---------IP---------
time    : 13*py^(1/2)*5^(1/2)/25 -> 97*py^(1/2)*5^(1/2)/125
y       : (810*py^(1/2)*5^(1/2)*t+(-(1261*py))+(-(625*t^2)))/125
y'      : (162*py^(1/2)*5^(1/2)+(-(250*t)))/25
y''     : -10

#---------4---------
---------PP---------
time    : 97*py^(1/2)*5^(1/2)/125
y       : 0
y'      : 128*py^(1/2)*5^(1/2)/125
y''     : UNDEF

---------IP---------
time    : 97*py^(1/2)*5^(1/2)/125 -> 6
y       : (27450*py^(1/2)*5^(1/2)*t+(-(59461*py))+(-(15625*t^2)))/3125
y'      : (1098*py^(1/2)*5^(1/2)+(-(1250*t)))/125
y''     : -10


#---------parameter condition---------
py      : [9, 11]

#time ended
```

図 7.5 bouncing_particle_interval の RCS による実行結果

```
INIT <=> x=0 & x'=1 .
EQ <=> [](x'' = 1  ).
BRAKE <=> [](x- >= 3 => x''= -x-).

INIT ,(EQ  << BRAKE).
```

図 7.6 ブレーキをかける車のモデル braking

過程において，ArcTan，つまり逆正接関数が出現することに起因する．5.1.3 節で示した通り，ArcTan をはじめとする逆三角関数に関しては，$[-\pi, \pi]$ という区間に変換をおこなっている．しかし，この区間があまりに精度が低いため，5.1.4 節で示した符号判定法の結果が Unknown になってしまっているためである．現在の実装では，符号判定の結果 Unknown が返った場合，5.1.3 節における $intPrec$ の値を増やす，つまり平方根に対するニュートン法を用いた区間値への変換をおこなう際の精度を上げた上で，大小判定を再試行するようにしている．しかし，今回の場合は区間の精度をどんなに上げたところで，逆三角関数に対しての区間値への変換処理は変わらないため，今回のようにいつまで経っても処理が終わらないという結果になっている．このように，三角関数および逆三角関数に関しては，その値の区間値への変換処理に問題があるため，より最適なものにする必要があると考えられる．

7.4 その他の例題

その他の HydLa プログラムの例題に関しても，RCS によりシミュレーション実行をおこない，その挙動を確認した．その結果，先述の bouncing_particle および bouncing_particle_interval を含めて，全部で 16 の例題において，正しいシミュレーションがおこなわれることを確認した．表 7.1 に，その一覧を示す．表中の○は，正しく実行が可能であることを意味し，△は条件によっては実行できない場合があることを意味する．

表からわかる通り，一部の例題に関しては，正しいシミュレーションをおこなうことができなかった．その例として，表中の dansa や box1 といった例題では，シミュレーション打ち切り時刻が小さければ問題ないが，打ち切り時刻を大きくすると正しいシミュレーションがおこなわれなくなるという問題が起きた．具体的には，計算の途中で突如 REDUCE 側から「+++ No space left at all」という文字列が Hyrose 側に送られ，実行が強制終了してしまうというものである．この現象は，先述の例題のように，ガード条

```
#---------1---------
---------PP---------
time    : 0
x       : 0
x'      : 1
x''     : 1

---------IP---------
time    : 0 -> -1+7^(1/2)
x       : 1/2*t*(2+t)
x'      : 1+t
x''     : 1

#---------2---------
---------PP---------
time    : -1+7^(1/2)
x       : 3
x'      : 7^(1/2)
x''     : -3

---------IP---------
time    : -1+7^(1/2) -> -1+7^(1/2)+2*ArcTan[1/3*7^(1/2)]
x       : 3*Cos[(1+(-1)*7^(1/2)+t)]+7^(1/2)*Sin[(1+(-1)*7^(1/2)+t)]
x'      : 7^(1/2)*Cos[(1+(-1)*7^(1/2)+t)]+(-3)*Sin[(1+(-1)*7^(1/2)+t)]
x''     : (-3)*Cos[(1+(-1)*7^(1/2)+t)]+(-1)*7^(1/2)*Sin[(1+(-1)*7^(1/2)+t)]

#---------3---------
---------PP---------
time    : -1+7^(1/2)+2*ArcTan[1/3*7^(1/2)]
x       : 3
x'      : (-1)*7^(1/2)
x''     : -3

---------IP---------
time    : -1+7^(1/2)+2*ArcTan[1/3*7^(1/2)] -> 4
x       : 1/2*(6+(1+(-1)*7^(1/2)+t+(-2)*ArcTan[1/3*7^(1/2)])^2
                +2*7^(1/2)*(-1+7^(1/2)+(-1)*t+2*ArcTan[1/3*7^(1/2)]))
x'      : 1+(-2)*7^(1/2)+t+(-2)*ArcTan[1/3*7^(1/2)]
x''     : 1


#time ended
```

図 7.7　braking の MCS による実行結果

（今回だけは `dump-in-progress` **オプションにより出力された結果を表示している）**

```
---------PP---------
time    : 0
x       : 0
x'      : 1
x''     : 1

---------IP---------
time    : 0 -> 7^(1/2)+(-1)
x       : (t^2+2*t)/2
x'      : t+1
x''     : 1

---------PP---------
time    : 7^(1/2)+(-1)
x       : 3
x'      : 7^(1/2)
x''     : -3
```

（以降，シミュレーション処理は終了せず）

図 7.8　braking の RCS による実行結果

件が複数ある例題や，無矛盾極大モジュール集合を求める過程で何度か矛盾を起こすような例題において，発生しやすいことが判明している．よって，こういった確認されている特徴から推測すると，シミュレーションをおこなう際に必要な計算がある一定量以上に達すると，前述のエラーメッセージの表示とともに強制終了してしまうのではないかと考えられる．

表 7.1　その他の例題に関しての実行結果

モデル名	モデルの説明	MCS での実行	RCS での実行
impulse_function	パルス関数	○	○
impulse_by_impluse	パルスをキャッチして上がるパルス	○	○
step_function	ステップ関数	○	○
sawtooth_wave	ノコギリ波	○	○
square_wave	方形波	○	○
triangle_wave	三角波	○	○
hypocycloid	内サイクロイドの軌跡	○	○
whirlpool	渦巻きの軌跡	○	○
dansa	段差をバウンドする質点	○	△
circle	円の中をバウンドする質点	○	○
ellipse	楕円の中をバウンドする質点	○	○
box1	箱がある空間を跳ね返る質点	○	△
bouncing_particle_rp_up	初期値に幅を持つ質点の投げ上げ	○	○
highway	走行する車 2 台中 1 台がブレーキ	○	△

第 8 章

まとめと今後の課題

8.1 まとめ

本研究により，REDUCE Constraint Solver は HydLa で記述されたモデルに対して，必要に応じて区間演算を交えつつ計算する高信頼な数式処理シミュレーションをおこなうことが可能となった．また，正しいシミュレーション結果が得られていることを，いくつかの例題をもとに確認した．

8.2 今後の課題

8.2.1 区間値への変換処理のクラス拡張

今後の課題としては，区間値への変換をおこなう対象のクラスを拡張することが挙げられる．現在は平方根など一部の無理数に対してしか区間値変換をおこなっていないため，三角関数や対数などの新たな無理数に対しても適切な区間値変換処理が求められる．

これにより，7.3 節で扱ったような，大小判定処理において三角関数が出現するようなモデルに対しても，適切な区間値によって包囲することで正しいシミュレーションがおこなえるようになると考えられる．

8.2.2 大小判定処理の最適化

5.1.3 節および 5.1.4 節にある通り，現在では無理数を含む定数式の大小判定をおこなう際，一旦無理数をある程度の精度の区間に変換し終わってから，初めて符号判定をおこなう．しかし，場合によっては区間の精度を上げるまでもなく，その判定結果が自明であ

るような定数式が存在することも確かである．そこで，区間値変換処理と符号判定処理とを改良し，両者を適切に協調しておこなうことによって，より効率的な，無駄のない処理に最適化することが可能であると考えられる．

8.2.3 複数の記号定数への対応

現段階の RCS においては，計算過程で出現する記号定数の個数は 1 個までという前提のもとで実装がおこなわれている．しかし，HydLa プログラム中の 2 つ以上の変数が初期値に幅を持つような場合などは当然考えられ，そのような場合，計算過程において記号定数が 2 個以上出現するため，現在の実装では正しい計算をおこなうことができないと考えられる．6.1.2 節で述べた，不等式タプルを用いた DNF による論理演算や，6.2.2 節で述べた，不等式の求解などにおける処理を拡張し，対応させる必要がある．

謝辞

まず，研究をおこなうにあたって，ご指導・助言をいただきました上田和紀教授には多大なる感謝の意を表します．また，HydLa 班の細部博史准教授には，ナント出張の際にたいへんお世話になり，また論文執筆にあたってはよくアドバイスをしていただきました．感謝致します．石井大輔氏には，私が B4 の頃から同じ HydLa 班メンバーとしてかわいがっていただき，時にはビデオ会議を使ってまでして研究の相談に乗っていただきました．感謝致します．HydLa 班の先輩である廣瀬賢一氏と大谷順司氏には，卒業されてからも合宿に来ていただいて論文執筆に協力していただくなど，大いにお世話になりました．感謝致します．そして同期の上田研 16 期の皆様には，研究生活の中で時には支え，時には支えられてここまでやってこられました．本当にありがとうございます．3 年間で色々なことがありましたね．kawabata さんは大吟醸熱燗だし，shibushun は海に物を奪われすぎだし，shimizu さんは実は最近ぱるる推しだし，nyakasen はトマトが食べられないし，yamane は麻雀と水道管ゲームが大好きだし，とても愉快で個性的な（？）メンバーに恵まれました．また，卒業旅行で行ったロシアも楽しかったですね．実は合宿などを除けば同期で行く初の旅行でしたが，最後にとても仲が深まったと思います．最終日の夜，モスクワで繰り広げられた大富豪は忘れられない思い出です．また，谷田部聡氏には，研究から就活，時には人生に関する相談にまで乗っていただくなど，たいへん助けられました．金沢でもがんばって．また餃子市にランチを食べに行こう．最後に，大学入学から今まで，いや，もっと前からずっと支え続けてくれた，両親と祖父母達に感謝を致します．

2012 年 2 月　高田　賢士郎

参考文献

[1] Borning, A., Feldman-Benson, B., Wilson, M. : Constraint Hierarchies, Lisp and Symbolic Computation, Vol.5, No.3(1992), pp. 223-270.

[2] Dolzmann, A., Seidl, A., Sturm, T. : Redlog User manual Edition 3.1, for redlog Version 3.06 (reduce 3.8), University of Passau, 2006.

[3] Granvilliers, L., Benhamou, F. : Algorithm 852: Realpaver: An Interval Solver using Constraint Satisfaction Techniques, ACM Trans. on Mathematical Software, Vol.32, 2006, pp. 138-156.

[4] Moore, R. E. : Interval Analysis, Prentice-Hall Englewood Cliffs, N.J., 1966.

[5] Wolfram Research, Mathematica, 技術・科学ソフトウェア.
http://www.wolfram.com/
http://reduce-algebra.com/

[6]

[7] 上田和紀, 石井大輔, 細部博史 : 制約概念に基づくハイブリッドシステムモデリング言語 HydLa, 第五回システム検証の科学技術シンポジウム, 2008, pp. 6-11.

[8] 上田和紀, 細部博史, 石井大輔 : ハイブリッド制約言語 HydLa の宣言的意味論, コンピュータソフトウェア, Vol.28, No.1, 2011.

[9] 大谷順司, 廣瀬賢一, 石井大輔, 上田和紀 : 不確実値を持つハイブリッドシステムの高信頼なシミュレーション手法, 第 6 回ディペンダブルシステムシンポジウム論文集, 2009, pp. 145-153.

[10] 佐々木優友 : 数式処理システム REDUCE によるハイブリッドシステムの実行とモデリング言語 HydLa への導入に向けた調査, 早稲田大学理工学部コンピュータ・ネットワーク工学科, 卒業論文, 2011.

[11] 渋谷俊, 高田賢士郎, 細部博史, 上田和紀 : ハイブリッドシステムモデリング言語 HydLa の実行アルゴリズム, コンピュータソフトウェア, Vol.28, No.3, 2011,

pp. 167-172.

[12] 廣瀬賢一, 大谷順司, 石井大輔, 細部博史, 上田和紀：制約階層によるハイブリッドシステムのモデリング手法, 日本ソフトウェア科学会第 26 回大会論文集, 2009, 2D-2.

[13] 松本翔太, 櫻庭翔, 高田賢士郎, 細部博史, 上田和紀： ハイブリッドシステムモデリング言語 HydLa の実装, 日本ソフトウェア科学会第 28 回大会, 講演論文集, 2011, pp. 306-311.

発表文献

[1] 高田賢士郎, 廣瀬賢一, 大谷順司, 石井大輔, 細部博史, 上田和紀： ハイブリッドシステムモデリング言語 HydLa の統合処理系, 第 12 回プログラミングおよびプログラミング言語ワークショップ, 2010.

[2] 渋谷俊, 高田賢士郎, 細部博史, 上田和紀： ハイブリッドシステムモデリング言語 HydLa 処理系の実行アルゴリズムの検討, 第 8 回ディペンダブルシステムワークショップ, 2010.

[3] 渋谷俊, 高田賢士郎, 細部博史, 上田和紀： ハイブリッドシステムモデリング言語 HydLa 処理系における実行アルゴリズム, 日本ソフトウェア科学会第 27 回大会, 講演論文集, 2010, 1B–2.

[4] 渋谷俊, 高田賢士郎, 細部博史, 上田和紀： ハイブリッドシステムモデリング言語 HydLa の実行アルゴリズム, コンピュータソフトウェア, Vol.28, No.3, 2011, pp. 167-172.

[5] 高田賢士郎, 渋谷俊, 細部博史, 上田和紀： ハイブリッドシステムモデリング言語 HydLa の数式処理実行系, 情報処理学会第 73 回全国大会, 2011, 1B–5.

[6] 松本翔太, 高田賢士郎, 細部博史, 上田和紀： ハイブリッドシステムモデリング言語 HydLa の処理系による非決定性の扱い, 第 13 回プログラミングおよびプログラミング言語ワークショップ, 2011.

[7] 松本翔太, 櫻庭翔, 高田賢士郎, 細部博史, 上田和紀： ハイブリッドシステムモデリング言語 HydLa の実装, 日本ソフトウェア科学会第 28 回大会, 講演論文集, 2011, pp. 306-311.

Appendix.A ソースコード

以下にソースコードを記載する.

- REDUCELink.h：68 行
- REDUCELink.cpp：104 行
- REDUCEStringSender.h：191 行
- REDUCEStringSender.cpp：404 行
- REDUCETime.h：17 行
- REDUCEValue.h：46 行
- REDUCEValue.cpp：54 行
- REDUCEVariable.h：17 行
- REDUCEVCS.h：99 行
- REDUCEVCS.cpp：390 行
- REDUCEVCSInterval.h：111 行
- REDUCEVCSInterval.cpp：569 行
- REDUCEVCSPoint.h：93 行
- REDUCEVCSPoint.cpp：368 行
- REDUCEVCSType.h：21 行
- SExpConverter.h：99 行
- SExpConverter.cpp：257 行
- vcs_reduce_source.h：9 行
- vcs_reduce_source.cpp：2755 行

で合計 5500 行程度のプログラムを記述した.